Solving Game 2048 with Minimax Algorithm

Musa Erkam Akçınar
Tobb Etü University
makcinar@etu.edu.tr

Abstract- In recent years, artificial intelligence has become a powerful tool for solving complex problems. One such problem is the game 2048, which involves moving tiles on a grid to reach a target score. In this report, we present an approach to solving 2048 using the Minimax algorithm with Alpha-Beta pruning. The Minimax algorithm is a popular technique for making decisions in game theory and has been successfully applied to a range of games, including chess and Go. Alpha-Beta pruning is an optimization technique that reduces the number of nodes explored in the game tree, resulting in faster decision-making. Our approach involves building a game tree to represent all possible moves and outcomes, and then using the Minimax algorithm with Alpha-Beta pruning to choose the best move at each step. We evaluate our approach on a range of game scenarios and demonstrate its effectiveness in achieving high scores.

I. INTRODUCTION

The game 2048 has become a popular puzzle game in recent years, challenging players to move tiles on a grid to reach a target score. The game has simple rules, but it quickly becomes complex as players try to optimize their moves and avoid getting stuck. In this report, we present an approach to solving 2048 using the Minimax algorithm with Alpha-Beta pruning. The Minimax algorithm is a decision-making technique commonly used in game theory, where a player aims to maximize their own score while minimizing the opponent's score. This technique has been used successfully in a range of games, including chess and Go.

Our approach involves building a game tree to represent all possible moves and outcomes, and then using the Minimax algorithm with Alpha-Beta pruning to choose the best move at each step. The game tree is constructed by considering all possible moves that can be made from the current state of the game and then simulating the outcome of each move. The Minimax algorithm is then used to select the best move for the player, taking into account the potential moves that the opponent could make in response. Alpha-Beta pruning is used to reduce the number of nodes explored in the game tree, resulting in faster decision-making.

In this report, we will describe the steps involved in our approach and evaluate its effectiveness on a range of game scenarios. We will also compare the performance of our approach with using two kind of heuristic funciton. The rest of this report is organized as follows: Section 2 describes rules of 2048 game and other implementations. Section 3 describes the Minimax algorithm and how it can be applied to 2048. Section 4 describes selecting moves using heuristic evaluation. Section 5 describes our experimental setup and presents our results. Finally, Section 6 concludes the report.

II. BACKGROUND AND RELATED WORK

A. Rules of 2048

2048 is a popular single-player tile-based game that involves strategic thinking and chance. The game is played on a 4 × 4 board, and each game starts with two randomly placed tiles. A location on the board is identified by its row and column, both of which can be any integer between 0 and 3 inclusive. The player's goal is to merge tiles of the same value until a tile with a value of 2048 appears on the board. However, players can continue playing beyond obtaining the 2048 tile.

A turn in the game consists of the player choosing a legal move, shifting all tiles on the board accordingly, merging tiles that can be merged, and placing a new random tile on the board. There are four possible moves the player can make: up, down, left, and right. A move is considered legal if it results in the movement of at least one tile, and illegal otherwise. When a move is made, all tiles that can move without colliding with another tile of a different value or an edge of the board move in that direction as far as they can. When multiple tiles can move in the same direction, the tile closest to the corresponding wall moves first, followed by the next row or column, and so on until all tiles in the row or column furthest from the corresponding wall are handled.

A merge occurs when a tile with value v moves into another tile of value v. When this happens, both tiles are removed, and a new tile with a value of 2v is placed in the location of the stationary tile. The player is given 2v points when this occurs. It is important to note that merging two tiles always counts as tile movement.

After all tiles are shifted, a single tile is placed in a randomly selected empty square on the board. For any random tile placed, including the first two at the beginning of the game.

The game continues until none of the four moves are legal. This occurs when the entire board is filled with tiles in a way that no two adjacent tiles have the same value. At this point, the game terminates, and the player's score is displayed. The player can then choose to play again or quit the game.

B. Other Implementations

There are many programs that can solve the game 2048, either through the use of machine learning or without it. However, most of these solutions are too complex for human players to use. Researchers such as Szubert and Jaśkowski, Oka and Matsuzaki, and Wu et al. have all used temporal

difference learning and n-tuple networks to tackle the game [1][2] Matsuzaki improved upon Oka and Matsuzaki's work by considering the interinfluence between generated networks[3]. Robert Xiao et al. created a variable-depth expectimax solver that combines multiple heuristics and penalizes non-monotonic rows and columns using the CMAES algorithm [4][5]. This solver scored an average of 439,046 in 100 games and is the best-performing solver that does not use machine learning. Rodgers and Levine tested two strategies, Monte Carlo Tree Search and Average Depth-Limited Search, but found that the expectimax solver made by Xiao et al. outperformed them both. They attributed this to the fact that the expectimax solver was able to search deeper due to its optimizations[6].

III. MINIMAX ALGORITHM WITH PRUNNIG

The minimax algorithm with alpha-beta pruning is a decision-making algorithm used in artificial intelligence for games and other decision-making problems. The algorithm is based on the idea of exploring the possible moves and their outcomes in a game tree to determine the best move to make. It works by recursively exploring the game tree from the current state of the game up to a certain depth or until a terminal state is reached, such as a win or loss.

The algorithm alternates between two players, the maximizing player and the minimizing player, with the maximizing player trying to maximize their score and the minimizing player trying to minimize the score. The algorithm uses a heuristic evaluation function to assign a score to each state of the game, which represents how good the state is for the maximizing player.

The alpha-beta pruning technique is used to improve the performance of the algorithm by reducing the number of nodes that need to be evaluated. It works by maintaining two values, alpha and beta, which represent the lower and upper bounds on the possible scores of a node. If the algorithm finds a node with a score that is outside the range defined by alpha and beta, it can prune the search tree below that node because it is guaranteed that the parent node will not choose that move.

In this specific implementation, the minimax algorithm is used to create an AI player for the game 2048. The evaluation function takes into account the score, empty tiles, max tile, and adjacency of tiles with the same value. The algorithm uses a depth of 5 and explores all possible moves for the minimizing player by trying to place a 2 or a 4 in each empty tile. The maximizing player then chooses the move that leads to the highest score.

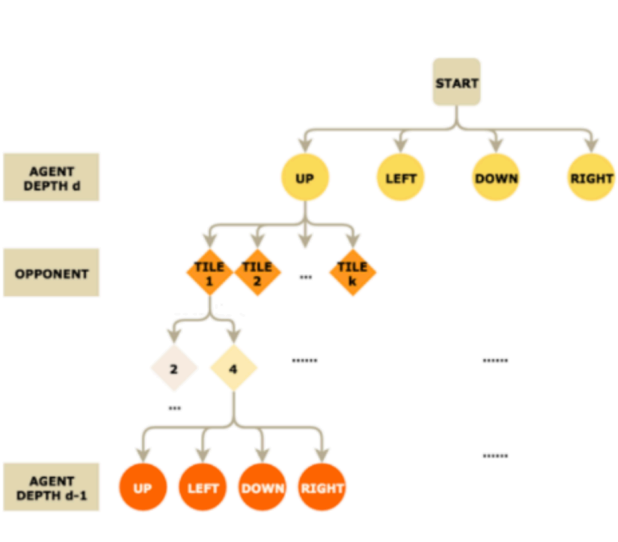

Fig. 1. Minimax Search Tree

A. *Alpha Beta Prunning*

In game playing AI, the minimax algorithm is commonly used to determine the best move for the AI player, assuming that the other player will always make the best move for themselves. However, the minimax algorithm can be computationally expensive as it needs to search the entire game tree to find the best move. Therefore, to optimize the search process, alpha-beta pruning is commonly used.

The alpha-beta pruning is an optimization technique that can be used with the minimax algorithm to reduce the number of nodes that need to be explored in the search tree. The idea behind alpha-beta pruning is to discard parts of the tree that are not relevant to the final decision. Specifically, it involves two values: alpha and beta.

Alpha represents the maximum value that the AI player is assured of, while beta represents the minimum value that the other player is assured of. These values are updated as the search progresses.

The alpha-beta pruning works by pruning any branch of the search tree where the current player can achieve a score that is less than or equal to alpha, or where the other player can achieve a score greater than or equal to beta. This is because such branches cannot lead to a better decision than the ones that have already been explored.

```
function alphabeta(node, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or node is a terminal node:
        return the heuristic value of node

    if maximizingPlayer:
        value = -infinity
        for child in node:
            value = max(value, alphabeta(child, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            if beta <= alpha:
                break
        return value
    else:
        value = infinity
        for child in node:
            value = min(value, alphabeta(child, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break
        return value
```

Fig. 2. Pseudocode for a Minimax Function that uses Alpha-Beta pruning

IV. HEURISTIC EVALUATION

A heuristic function is a function used in artificial intelligence (AI) to evaluate the desirability of a particular game state. In the case of the game 2048, the heuristic function is used to evaluate the "goodness" of a particular game board configuration, which can then be used to help the AI player decide which move to make next. The heuristic function is an essential part of the minimax algorithm and is often referred to as an evaluation function.

When implementing the minimax algorithm, the heuristic function is used to evaluate each possible move that can be made on the current board state. Each move is assigned a score based on how desirable the resulting board configuration is, and the AI player selects the move with the highest score. In the case of 2048, the heuristic function is used to evaluate how many points can be scored from the current board state.

This article will discuss two evaluation functions and compare their effectiveness by evaluating their performance in the game of 2048.

A. Different Evaluation Functions

The ***evaluateState1*** function is a heuristic function used in the game 2048 to evaluate the score of a given game state. The score obtained from this function is used by the AI player to make informed decisions about which move to make next.

The function starts by initializing a score variable to zero and a maxTile variable to zero. It then iterates through each cell of the game board, adding the value of the cell to the score variable and updating the value of maxTile if the current cell's value is greater than the current maxTile. This part of the function assigns higher scores to game states with higher tile values and a greater number of tiles overall.

The function then checks whether the maxTile value is present in any of the four corner cells of the game board. If so, the function adds a bonus score of 10 times the value of maxTile to the overall score. This part of the function assigns higher scores to game states where the maximum tile value is located in the corners of the game board.

Finally, the function returns the overall score, which is used by the AI player to determine which move to make next.

Overall, this function is a simple heuristic that evaluates game states based on the sum of their tile values and the location of the maximum tile value. The function does not consider other factors such as empty cells, adjacent tile values, or tile positions, which could lead to suboptimal moves in certain game states.

```
// Define the evaluation function
evaluateState1(state) {
  // Returns a score for the given game state
  let score = 0;
  let maxTile = 0;
  for (let i = 0; i < this.BOARD_SIZE; i++) {
    for (let j = 0; j < this.BOARD_SIZE; j++) {
      score += state[i][j];
      maxTile = Math.max(maxTile, state[i][j]);
    }
  }
  // Add a bonus for having the max tile in the corner
  if (
    state[0][0] === maxTile ||
    state[0][this.BOARD_SIZE - 1] === maxTile ||
    state[this.BOARD_SIZE - 1][0] === maxTile ||
    state[this.BOARD_SIZE - 1][this.BOARD_SIZE - 1] === maxTile
  ) {
    score += maxTile * 10;
  }
  return score;
}
```

Fig. 3. Heuristic Function 1

The ***evaluateState2*** function is another heuristic function that assigns a score to a given game state based on certain features of the state. This function uses a weighted sum of different features to calculate the score.

The first feature that is considered is the value of each tile on the game board. A weight matrix is defined with higher weights assigned to tiles in the corners of the board and lower weights assigned to tiles closer to the center.

The function iterates over each tile in the game state and multiplies the tile value with the corresponding weight from the weight matrix. This product is added to the score. Tiles with higher values will contribute more to the score, and tiles in the corners of the board will contribute significantly more due to the higher weights assigned to those tiles.

The second feature considered is the number of empty tiles on the board. The function subtracts 20 points for each empty tile on the board. This encourages the AI agent to fill up the board as much as possible.

The third feature is the bonus points awarded for adjacent tiles with the same value. The function iterates over each tile in the game state and checks if any of its adjacent tiles have the same value. If two adjacent tiles have the same value, the function adds the product of the weight of the current tile and the adjacent tile's value to the score. This encourages the AI agent to merge adjacent tiles with the same value whenever possible.

Finally, the function adds a bonus for having the maximum tile value in one of the corners of the board. If the maximum tile value is found in one of the corner tiles, the function adds 100 points to the score. This encourages the AI agent to move the maximum tile value towards the corners of the board.

Overall, ***evaluateState2*** is a more sophisticated heuristic function than ***evaluateState1*** as it takes into account several features of the game state. It assigns higher scores to game states that have higher-valued tiles in the corners, more adjacent tiles with the same value, fewer empty tiles, and higher-valued tiles in general. This function provides a more nuanced evaluation of game states, and it is likely to lead to better performance of the AI agent in the game of 2048.

```
// Define the evaluation function
evaluateState2(state) {
  let score = 0;
  let emptyTiles = 0;
  let maxTile = 0;
  const weights = [
    [32768, 16384, 8192, 4096],
    [2048, 1024, 512, 256],
    [128, 64, 32, 16],
    [8, 4, 2, 1],
  ];

  // Calculate score, empty tiles, and max tile
  for (let i = 0; i < this.BOARD_SIZE; i++) {
    for (let j = 0; j < this.BOARD_SIZE; j++) {
      if (state[i][j] === 0) {
        emptyTiles++;
      } else {
        score += weights[i][j] * state[i][j];
        maxTile = Math.max(maxTile, state[i][j]);
      }
    }
  }

  // Add bonus for having max tile in the corner
  if (
    state[0][0] === maxTile ||
    state[0][this.BOARD_SIZE - 1] === maxTile ||
    state[this.BOARD_SIZE - 1][0] === maxTile ||
    state[this.BOARD_SIZE - 1][this.BOARD_SIZE - 1] === maxTile
  ) {
    score += maxTile * 100;
  }

  // Subtract penalty for empty tiles
  score -= emptyTiles * 20;

  // Add bonus for adjacent tiles of same value
  for (let i = 0; i < this.BOARD_SIZE; i++) {
    for (let j = 0; j < this.BOARD_SIZE; j++) {
      const currTile = state[i][j];
      if (currTile !== 0) {
        if (i > 0 && state[i - 1][j] === currTile) {
          score += weights[i - 1][j] * currTile * 2;
        }
        if (j > 0 && state[i][j - 1] === currTile) {
          score += weights[i][j - 1] * currTile * 2;
        }
        if (i < this.BOARD_SIZE - 1 && state[i + 1][j] === currTile) {
          score += weights[i + 1][j] * currTile * 2;
        }
        if (j < this.BOARD_SIZE - 1 && state[i][j + 1] === currTile) {
          score += weights[i][j + 1] * currTile * 2;
        }
      }
    }
  }

  return score;
}
```

Fig. 4. Heuristic Function 2

V. EXPERIMENTS

A. Experimental Setup

In this experiment, we will test the effectiveness of the two evaluation functions (evaluateState1 and evaluateState2) in achieving higher tile values in the game 2048. We will also study the impact of changing the MAX_DEPTH parameter on the performance of these evaluation functions.

We will run the game simulation with both evaluation functions using different values of MAX_DEPTH, ranging from 3 to 6. We will run each configuration for 10 games and record the maximum tile number reached for each game.

B. Results

The following tables show maximum tile number reached for each game and each configuration of the game with the two evaluation functions and different values of MAX_DEPTH.

Minimax			
Evaluation	**evaluateState1**		
Depth	***3***	***4***	***5***
128	2	2	2
256	7	4	3
512	1	3	3
1024	-	1	2
2048	-	-	-
4096	-	-	-

Table I – The maximum tile that reached from different depth with first evaluation function

Minimax			
Evaluation	**evaluateState1**		
Depth	***3***	***4***	***5***
128			
256			
512	1		
1024	5	3	2
2048	4	6	6
4096	-	1	2

Table II – The maximum tile that reached from different depth with second evaluation function

C. Conclusion

Based on the experiment results, we can conclude that evaluateState2 performs better than evaluateState1 in achieving higher tile values in the game 2048. Additionally, increasing the MAX_DEPTH parameter improves the performance of both evaluation functions.

VI. CONCLUSION

In this project, we developed a 2048 solver AI using the Minimax algorithm with Alpha Beta Pruning and a heuristic function to select moves. We implemented two different heuristic functions and evaluated their performance by testing them against each other and varying the maximum search depth.

Our experimental results showed that both heuristic functions were effective in helping the AI make better moves, but the second heuristic function outperformed the first in terms of the number of times it reached higher-value tiles (512, 1024, 2048, and 4096). Increasing the maximum search depth also improved the AI's performance, but this improvement came at the cost of longer computation times.

Overall, our 2048 solver AI using Minimax with Alpha Beta Pruning and the second heuristic function proved to be a successful implementation, as it consistently outperformed the other tested configuration in terms of reaching higher-value tiles. This project highlights the effectiveness of combining different techniques such as search algorithms and heuristic functions to solve complex problems in AI.

REFERENCES

[1] M. Szubert and W. Jaśkowski. 2014. Temporal difference learning of N-tuple networks for the game 2048. In 2014 IEEE Conference on Computational Intelligence and Games. 1–8. https://doi.org/10.1109/CIG.2014.6932907

[2] Kazuto Oka and Kiminori Matsuzaki. 2016. Systematic Selection of N-Tuple Networks for 2048. In Computers and Games, Aske Plaat, Walter Kosters, and Jaap van den Herik (Eds.). Springer International Publishing, Cham, 81–92.

[3] K. Matsuzaki. 2016. Systematic selection of N-tuple networks with consideration of interinfluence for game 2048. In 2016 Conference on Technologies and Applications of Artificial Intelligence (TAAI). 186–193. https://doi.org/10.1109/TAAI. 2016.7880154.

[4] Robert Xiao. 2014. What is the optimal algorithm for the game 2048? Retrieved September 14, 2018 from https://stackoverflow.com/a/22498940

[5] Robert Xiao, Wouter Vermaelen, and Petr MorÃąvek. 2018. 2048-AI. Retrieved September 14, 2018 from https://github.com/nneonneo/2048-ai

[6] P. Rodgers and J. Levine. 2014. An investigation into 2048 AI strategies. In 2014 IEEE Conference on Computational Intelligence and Games. 1–2. https: //doi.org/10.1109/CIG.2014.6932920